Introduction to UML and Design Patterns

DF14900 Software Engineering
CUGS
2011

Christoph Keusler and Kristian Sandahl
Department of Computer and Information Science
Linköping University, Sweden

Oak library

Part I
Modeling Structure:
Classes and Objects

Part II
Short Introduction to Design Patterns

Part III
Modeling Behavior:
State Machines etc.

The goals of module design

§ Provide the expected function
§ Prepare for change:
  Separation of concern
  Testability
  Understandability
§ Contribute to quality, eg:
  Performance
  Usability
  Reliability
  ...

A Software Life-cycle Model…
Which part will we talk about today?

Agenda

Part I:
Structural Modeling with UML
Esp., Classes and Objects

Part II:
Short Introduction to Design Patterns

Part III:
Behavioral Modeling with UML
State Machines, Sequence Diagrams,
Use case diagrams, Activity Diagrams

Modeling as a Design Technique

§ Testing a physical entity before building it
§ Communication with customers
§ Visualization
§ Reduction of complexity
§ Models supplement natural language
§ Models support understanding, design, documentation
§ Creating a model forces you to take necessary design decisions
§ UML is now the standard notation for modeling software.
**Literature on UML**

- Current version is UML 2.0 (2004/2005)
- OMG documents: UML Infrastructure, UML Superstructure
- Books:
  - Pfleeger: Software Engineering 3rd ed., 2005 (mostly Chapter 6)
  - And many others…

**UML: Different diagram types for different views of software**

- Modeling (logical) structure of software:
  - Static view: Class diagram
  - Design view: Structure diag., collaboration d., component d.
  - Use case view: Use case diagram

- Modeling behavior of software:
  - Activity view: Activity diagram
  - State machine view: State machine diagram
  - Interaction view: Sequence diagram, communication diagram

- Modeling physical structure of software
  - Deployment view: Deployment diagram

- Modeling the model, and extending UML itself
  - Model management view: Package Diagram
  - Profiles

**A Single Class**

- Class name
- attributes
  - name: String[1]
  - email: String[0..2]
- operations
  - getUsername(): Integer
  - setUsername(String)
  - addEmail(email: String)
- visibility
  - + public
  - - private
  - # protected
  - ~ package
- Multiplicity
  - 1 exactly one
  - 0..1 Zero or one
  - * Zero or more
  - (same as 0..*)
  - 2..8 Between 2 and 8

**Relationships (1/6) - overview and intuition**

- Association
- Both representations are almost equivalent
- navigation is not allowed

**Relationships (1/6) - overview and intuition**

- Both representations are almost equivalent
- Navigation - mycar can reach the wheels, but not the opposite
- Explicitly show that navigation is not allowed

**Part I**
Modeling Structure: Classes and Objects

**Part II**
Short Introduction to Design Patterns

**Part III**
Modeling Behavior: State Machines etc.
Relationships (1/6) - overview and intuition
- Association

What does it mean to have a * here? What if we have multiplicity 1 instead?

A wheel can only be linked to one car instance.

A wheel can be linked to more than one car instance.

Part I Modeling Structure: Classes and Objects
Part II Short Introduction to Design Patterns
Part III Modeling Behavior: State Machines etc.

Relationships (2/6) - overview and intuition
- Aggregation

“A” has a reference(s) to instance(s) of “B”. Alternative: attributes

Aggregation

Part I Modeling Structure: Classes and Objects
Part II Short Introduction to Design Patterns
Part III Modeling Behavior: State Machines etc.

Relationships (3/6) - overview and intuition
- Composition

“A” has a reference(s) to instance(s) of “B”. Alternative: attributes

Composition

Part I Modeling Structure: Classes and Objects
Part II Short Introduction to Design Patterns
Part III Modeling Behavior: State Machines etc.

Relationships (4/6) - overview and intuition
- Association

Associations are the “glue” that ties a system together

association instance = link

An association describes a relation between objects at run-time.

{ (mycar1, wheel1), (mycar1, wheel2), (mycar1, wheel3), (mycar1, wheel4) }

Part I Modeling Structure: Classes and Objects
Part II Short Introduction to Design Patterns
Part III Modeling Behavior: State Machines etc.

Relationships (5/6) - overview and intuition
- Aggregation

A major source of confusion

Common vague interpretations: “owns a” or “part of”

Aggregation was added to UML with little semantics. Why?

Jim Rumbaugh

“Think of it as a modeling placebo”

Recommendation: - Do not use it in your models.
- If you see it in others’ models, ask them what they actually mean.

Part I Modeling Structure: Classes and Objects
Part II Short Introduction to Design Patterns
Part III Modeling Behavior: State Machines etc.

Relationships (6/6) - overview and intuition
- Composition

Yes! First, multiplicity must be 1 or 0..1. An instance can only have one owner.

But, isn’t this equivalent to what we showed with associations?

Well, in this case...

Part I Modeling Structure: Classes and Objects
Part II Short Introduction to Design Patterns
Part III Modeling Behavior: State Machines etc.
**Part I: Modeling Structure - Classes and Objects**

- Generalization Relationships (4/6) - overview and intuition
- Composition Relationships (3/6) - overview and intuition

**Part II: Short Introduction to Design Patterns**

- Realization
- Generalization

**Part III: Modeling Behavior: State Machines etc.**

- Composition
- Generalization

---

**Class with code for the drive operation**

```
Class Vehicle

// Inheritance
Vehicle => Car

// Overriding
Car drive()
Motorcycle reverse()
```

---

**Key concepts**

- "No sharing" rule
- This owner is responsible for managing its parts, e.g., allocation and deallocation.

---

**Using composition...**

- **Car**
  - Wheel
  - MotorCycle

**Using associations...**

- **Car**
  - Wheel
  - MotorCycle

---

**Can mycart and mybike1 share the same wheels?**

**No!** Not with composition!

Key concepts:
- "No sharing" rule
- This owner is responsible for managing its parts, e.g., allocation and deallocation.

---

**Composition**

An instance of "B" is part of an instance of "A", where the former is not allowed to be shared.

**Generalization**

1) "A" inherits all properties and operations of "B".
2) An instance of "A" can be used where a specific type is expected.

---

**Using composition...**

- **Car**
  - Wheel
  - MotorCycle

---

**Can mycart and mybike1 share the same wheels?**

**No!** Not with composition!
What is the difference between an interface and an abstract class?

- Interface:
  - Provides a specified interface
  - Must implement the interface

- Abstract class:
  - Cannot contain implementation
  - Non of them can be instantiated
  - Can (but need not to) contain implementation

An abstract class with only abstract operations is conceptually the same as an interface.

What is the difference between an interface and an abstract class?

- Interface:
  - Provides a specified interface

- Abstract class:
  - Cannot contain implementation

Can we create an instance of Vehicle? Real it is a concrete.

Door

AnotherVehicle

Vehicle

Can we create an instance of AnotherVehicle? Not!

Door

AnotherVehicle

Vehicle

Restriction

Abstraction

Realization

Composition

Aggregation

Association

Generalization

Realization

Dependency

Cannot contain implementation

Abstract class

A

B

Association

"A" has a reference(s) to instance(s) of "B". Alternative: attributes

A

B

Aggregation

Avoid it to avoid misunderstandings

A

B

Composition

An instance of "B" is part of an instance of "A", where the former is not allowed to be shared.

A

B

Generalization

1) "A" inherits all properties and operations of "B"
2) An instance of "A" can be used where a instance of "B" is expected.

A

B

Realization

"A" provides an implementation of the interface specified by "B".

A

B

Dependency

Forward engineering can be very useful to see dependencies between classes and modules!

A

B

Aggregation

Avoid it to avoid misunderstandings

A

B

Composition

An instance of "B" is part of an instance of "A", where the former is not allowed to be shared.

A

B

Generalization

1) "A" inherits all properties and operations of "B"
2) An instance of "A" can be used where a instance of "B" is expected.

A

B

Realization

"A" provides an implementation of the interface specified by "B".

A

B

Dependency

"A" is dependent on "B" if changes in the definition of "B" causes changes of "A".

A

B

Association

"A" has a reference(s) to instance(s) of "B". Alternative: attributes
### Part I: Modeling Structure

#### Classes and Objects

#### Class model with inheritance and abstract classes

- **CoffeeCustomer**
  - `pay(c: coin)`
  - `getCan()`
- **IndividualCustomer**
  - `getCan()`
- **Porter**
  - `getCan()`

**Abstract class** (cannot be instantiated, only extended/specialized)

`pay()` method is inherited from `CoffeeCustomer`.

#### Classes and objects

**Classes:**
- `CoffeeCustomer`
- `IndividualCustomer`
- `Porter`
- `CupOfCoffee`

**Objects:**
- `Kristian: CoffeeCustomer`
- `c1: CupOfCoffee`
- `c2: CupOfCoffee`

**Reasoning about an arbitrary object**

Like this:

```
<#CoffeeCustomer> buyc <#CupOfCoffee>
```

...or simply like this:

```
<#CoffeeCustomer> buyc <#CupOfCoffee>
```

### Part II: Short Introduction to Design Patterns

**Christopher Alexander**

**TAPESTRY OF LIGHT AND DARK**

Create alternating areas of light and dark throughout the building, in such a way that *people naturally walk toward the light*, whenever they are going to important places: seats, entrances, stairs, passages, places of special beauty, and make other areas darker, to increase the contrast.
Software Design Patterns

A Design Pattern is a standard solution for a standard design problem in a certain context.

Goal: reuse design information

"A pattern involves a general description of a recurring solution to a recurring problem with various goals and constraints. It identifies more than a solution, it also explains why the solution is needed." (James Coplien)

"... describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" (Christopher Alexander)

Example: Facade

Pattern Catalog in Gamma et al. 1996:

The GoF book describes each pattern using the following attributes:
The name to describes the pattern, its solutions and consequences in a word or two
The problem describes when to apply the pattern: intent, motivation, applicability
The solution describes the elements that make up the design (structure with participants), their relationships, responsibilities, and collaborations/interactions
The consequences are the results and trade-offs in applying the pattern
Also: implementation notes, known uses, related patterns.
All examples in C++ and Smalltalk

Remark: Patterns exist also beyond the OO world…
Facade

Intent
Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Motivation
Structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems.

... example ...

Facade

Applicability
Use the Facade pattern when:

1. you want to provide a simple interface to a complex subsystem. This makes subsystems more reusable and easier to customize.
2. there are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from other subsystems, thereby promoting subsystem independence and portability.
3. you want to layer your subsystems. Use a facade to define an entry point to each subsystem level.

Facade

Consequences
The Facade pattern offers the following benefits:

1. It shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making subsystem easier to use.
2. It promotes weak coupling between subsystem and its clients. Weak coupling lets you vary the components of the subsystem without affecting its clients.
3. It doesn’t prevent applications from using subsystem classes if they need to.

Observer

Applicability

When an abstraction has two aspects, one dependent on the other.
When a change to one object requires changing others.
When an object should be able to notify other objects without making assumptions about who these objects are.

Observer, structure

Subject

attach(observer)
detach(observer)
notify()

ConcreteSubject

subjectState
getState()
setState()

Observer

update()

ConcreteObserver

observerState
update()
Observer, collaborations

Observer, consequences

Strategy

Design Pattern Space [Gamma et al.'96]

Part III
Modeling Behavior in UML:
State Machines etc.
Further UML Features
State machine diagram

For class CoinHandler:
- Checking state
- Idle state
- Transition:
  - falseCoin(): returnCoin(self)
  - insertCoin(): checkCoin(self)
- Start state marker
- Trigger event, causing transition
- Action, reaction to the event

Orthogonal, composite state

Course attempt state machine
- Studying
  - Lab 1
  - Lab 2
- Project
- Final exam
- Project done
- Final exam pass
- Failed
- Passed

Explicit exit points

Course attempt:
- Lab 1 done
- Lab 2 done
- Project
- Final exam
- Failed
- Passed

Sequence diagram

Sequence diagram with several objects

Combining fragments of interaction diagrams

SD processOrder
- Order
- TicketDB
- Account
- Ref
- Get existing customer data
- Loop
- Get next item
- Reserve(date, no)
- Add(seats)
- Destroy
- Loop condition
- Answer
Use-case modelling

A use-case is:

“... a particular form or pattern or exemplar of usage, a scenario that begins with some user of the system initiating some transaction of sequence of interrelated events.”

Jacobson et al. 1992: Object-oriented software engineering. Addison-Wesley

Use-case diagram for the coffee machine

A CoffeeDrinker approaches the machine with her cup and a coin of SEK 5. She places the cup on the shelf just under the pipe. She then inserts the coin, and presses the button for coffee to get coffee according to default settings. Optionally she might use other buttons to adjust the strength and decide to add sugar and/or whitener. The machine processes the coffee and rings a bell when it is ready. The CoffeeDrinker takes her cup from the shelf.

Identifying classes: noun analysis

A CoffeeDrinker approaches the machine with her cup and a coin of SEK 5. She places the cup on the shelf just under the pipe. She then inserts the coin, and presses the button for coffee to get coffee according to default settings. Optionally, she might use other buttons to adjust the strength and decide to add sugar and/or whitener. The machine processes the coffee and rings a bell when it is ready. The CoffeeDrinker takes her cup from the shelf.
### Extended class model

- **CoffeeCustomer**
- **CupOfCoffee**
- **Porter**
- **CanOfCoffee**

### Revised class model

- **CoffeeCustomer**
- **CupOfCoffee**
- **Porter**
- **CanOfCoffee**

### The coffee machine class model

- **Interface**
- **CoinHandler**
- **Brewer**

### Communication diagram

- **Interface**
- **CoffeeCustomer**
- **CoinHandler**
- **Brewer**

### Activity Diagram

1. **Graph**
   - **Nodes are activities** (actions)
     - Method invocations, operations, sending / receiving messages, handling events, creating / accessing / modifying / deleting objects, variables ...
     - Data flow by input and output parameter pins
   - **Edges are control flow transitions**
     - To some degree dual to the state diagram
   - **Might be refined to a low-level specification;**
     - cf. control flow graph (~ compiler IR)
   - **A Petri Net**
     - Interpretation by moving tokens along edges
     - Models concurrency by multiple tokens for “current state”
     - Fork / join for synchronization
   - **Models real-world workflows**

2. **Activity diagram**

   - **Initial node**
   - **Final node**
   - **Decision**
   - **Join**
   - **Fork**
   - **Add hot water to adjust strength**
   - **Pour coffee**

---

**Even small models take space. You need good drawing tools and a large sheet.**

**Shows message flows with sequence numbers**

**Similar information as sequence diagram**

**Brew coffee**

---

**Insert coin**

**Coin accepted?**

**Fork**

**Join**

---

**Part I**
- **Modeling Structure:** Classes and Objects

**Part II**
- **Short Introduction to Design Patterns**

**Part III**
- **Modeling Behavior:** State Machines etc.
Other UML features...

- **Comments**
- **Constraints** in OCL (Object Constraint Language)

![Constraint diagram](image)

- **Profiles**: Collections of stereotypes for specific domains, e.g. Realtime-profile for UML
  - Customize (specialize) UML elements, e.g. associations
  - Can introduce own symbols
  - More in the lecture on Model-Driven Architecture...

- **MOF** (Meta-Object Facility):
  - UML is specified in UML MOF, a core subset of UML
  - MOF is the meta-model of UML – a language to define UML
  - Powerful mechanism for extending UML by adding new language elements

Homework Exercise

- Draw a class diagram for the following scenario:

  A customer, characterized by his/her name and phone number, may purchase reservations of tickets for a performance of a show. A reservation of tickets, annotated with the reservation date, can be either a reservation by subscription, in which case it is characterized by a subscription series number, or an individual reservation. A subscription series comprehends at least 3 and at most 6 tickets; an individual reservation at most one ticket. Every ticket is part of a subscription series or an individual reservation, but not both. Customers may have many reservations, but each reservation is owned by exactly one customer. Tickets may be available or not, and one may sell or exchange them. A ticket is associated with one specific seat in a specific performance, given by date and time, of a show, which is characterized by its name. A show may have several performances.